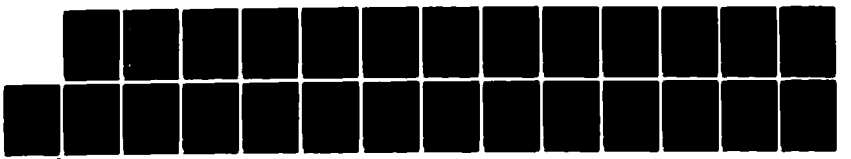
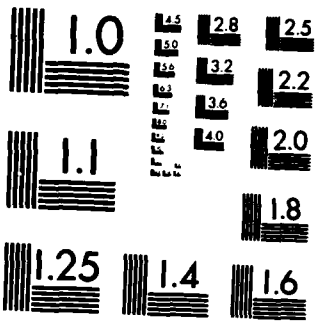
AD-A141 026 PARALLEL PROGRAMMING AND THE POKER PROGRAMMING ENVIRONMENT(U) WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE L SNYDER APR 84 TR-84-04-02 N00014-80-K-0816 1/1

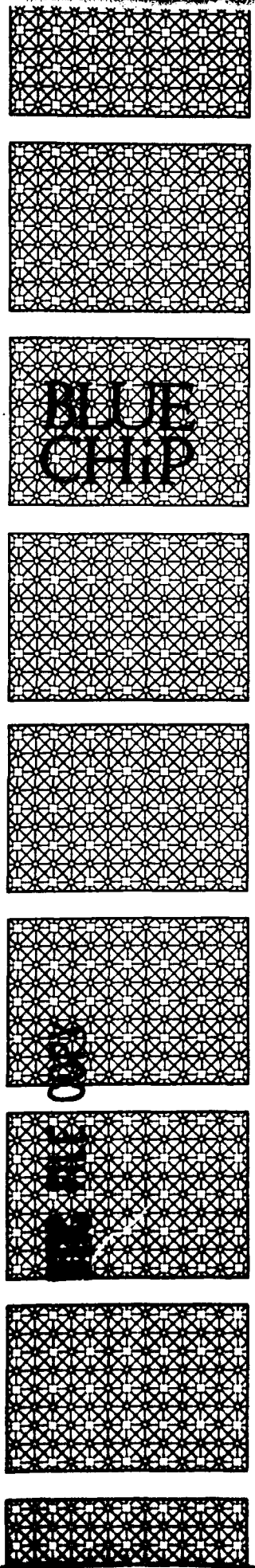
UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
6-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



(12)

Parallel Programming and the Poker Programming Environment

by

Lawrence Snyder

The BLUE CHiP Project
University of Washington
Department of Computer Science, FR-35
Seattle, Washington 98195

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-84-04-02	2. GOVT ACCESSION NO. AD-A144 026	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PARALLEL PROGRAMMING AND THE POKER PROGRAMMING ENVIRONMENT		5. TYPE OF REPORT & PERIOD COVERED Technical, interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Lawrence Snyder		8. CONTRACT OR GRANT NUMBER(s) N00014-80-K-0816 N00014-81-K-0360 N00014-84-K-0143
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science, FR-35 Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task SRO-100
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, Virginia 22217		12. REPORT DATE April 1984 13. NUMBER OF PAGES 22
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

programming environment, CHiP Computer, Pringle computer, programming language , programming, interactive environment, XX programming language, graphics programming language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Parallel programming is described as the conversion of an abstract, machine independent algorithm to a form, called a program, suitable for execution on a particular computer. The conversion activity is simplified where the form of the abstraction is close to the form required of the programming system. Fine mechanisms are identified as commonly occurring in algorithms specification. The Poker Parallel Programming Environment is known to support these five mechanisms conveniently; thus the conversion is easy and

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

the parallel programming is simple. The Poker environment is described and examples are provided. An analysis of the efficiency of the programming facilities provided by Poker is given and they all seem to be very efficient.

Parallel Programming and the Poker Programming Environment

Lawrence Snyder
Department of Computer Science
University of Washington

TR-84-04-02

ABSTRACT

Parallel programming is described as the conversion of an abstract, machine independent algorithm to a form, called a program, suitable for execution on a particular computer. The conversion activity is simplified where the form of the abstraction is close to the form required of the programming system. Five mechanisms are identified as commonly occurring in algorithms specification. The Poker Parallel Programming Environment is known to support these five mechanisms conveniently; thus the conversion is easy and the parallel programming is simple. The Poker environment is described and examples are provided. An analysis of the efficiency of the programming facilities provided by Poker is given and they all seem to be very efficient.

The Poker effort is part of the Blue CHiP Project which has been funded by the Office on Naval Research under Contracts N00014-80-K-0816, N00014-81-K-0360 (SRO-100) and N00014-84-K-0143.

Parallel Programming and the Poker Programming Environment

Lawrence Snyder
Department of Computer Science
University of Washington

Introduction

The number of parallel computers that exist only as paper designs greatly exceeds the number that have been built. The number of parallel computers that have been built greatly exceeds the number that have become stable enough to go into productive use. A machine in productive use implies the existance of a programmer population, but because parallel computers are often unique or one of only a few copies, the population tends to be small. (In fact it was possible[1] to track down virtually everyone who ever programmed the Illiac IV!) Obviously, it is a rare individual who has written and *run* a parallel program.

Many of us may one day join this small, select group of programmers, however, as parallel computers become more widely available in response to recently recognized critical needs.[2,3] It is, therefore, natural to wonder what parallel programming is like and how it differs from the familiar sequential programming. Our answers will be reassuring in that although we show parallel programming to be quite different, it is nevertheless straightforward and understandable. Our approach is to begin at the beginning and establish *what* the programmer must accomplish in parallel programming. Then, after establishing the given conditions, we analyze *how* it might be done in a particular parallel programming environment.

A parallel programming environment is the collection of all language and operating system facilities needed to support parallel programming. We give an overview of the Poker Parallel Programming Environment which has been developed to support the CHiP Computer.[4] (No knowledge of the CHiP Computer is presumed.) The Poker environment runs on a "frontend" sequential computer (VAX 11/780) and serves as a comprehensive system for writing and running parallel programs. It is sufficiently general that, with minor modification, it could be a parallel programming environment for any of a half dozen recently proposed ensemble parallel computers.[15]

The Parallel Programming Activity

Before building a parallel programming environment, that is, a system with a complete set of language and support facilities for parallel programming, one must scrutinize the programming activity, searching for those things that can be included to help make it easy, and searching for those things that must be excluded to avoid making it hard. This scrutiny, as will soon become apparent, leads one to examine how parallel algorithms are specified in the technical literature. But first, what do programmers do exactly?

Programming, either sequential or parallel, is the conversion of an abstract (machine independent) algorithm into a form, called a *program*, that can be run on a particular computer. The algorithm is an abstraction describing a process that could be implemented on many machines. The program is an implementation of the algorithm for a particular machine. Programming is the conversion activity. Since it is a conversion activity, programming will be easy or difficult depending on whether the algorithmic form is similar or dissimilar to the desired program form. But what are the sources of dissimilarity between algorithm and program?

First, algorithms are abstractions whose generality is intended to transcend the specifics of any implementation. Thus, when an algorithm is specified in the technical literature, there are many details purposely omitted, or at best merely implied, because they have little or no bearing on the operation of the algorithm. These must be made explicit in the course of programming, since they must be defined by the time the program is executed. There seems not to be much point (or much possibility) trying to develop a software support system to reduce this source of dissimilarity. It is inherent.

The second source of dissimilarity is a mismatch of mechanisms between those used in the algorithm specification and those provided by the programming system. For a sequential programming example of this phenomenon, consider the mechanism of recursion and imagine programming a recursive algorithm in a nonrecursive programming language. The programming is difficult because one must, in effect, implement a support package for recursion within the existing mechanisms of the language. A programming environment will reduce dissimilarity due to mechanism mismatch when the form required of its programs is similar to the form the algorithms already have, i.e. when there is a minimum amount of conversion to be done. Thus, this source of dissimilarity is not inherent; it can be removed.

The ideal programming environment, then, cannot make parallel programming effortless, since there will always be some dissimilarity due to the inherent properties of abstraction. It could greatly simplify the programming task, however, by supporting a specificational form *close* to that used to give algorithms in the technical literature. Although this might appear to be an unattainable goal, since algorithms are given in the literature in a form unencumbered by any preordained syntax or semantics, and are intended for thinking readers rather than computers, it happens that common characteristics of parallel algorithm specification can be identified. From these properties, parallel programming mechanisms can be developed.

In order to illustrate the common characteristics of a parallel algorithm specification, we begin by giving two parallel algorithms.

- *Example I.* Kung and Leiserson[5] describe their systolic band-matrix multiplication algorithm with the picture shown in Figure 1 together with the explanation that each processor repeatedly executes a three step cycle, two of which are idle steps and the third is an "inner product" step defined by the text

  read A, B, C
  C ← C + AB
  write A, B, C

  such that all processors of every third (horizontal) row execute their inner product step simultaneously while the others are idle. The A band-matrix enters through the upper left edge, the B band-matrix enters through the upper right edge, and the result is emitted along the top edge.

- *Example II.* Schwartz[6] presents an algorithm in which the maximum of n log n values is found in time proportional to log n using n processes connected together in a complete binary tree, provided that initially each process has log n of the values; all processes begin by finding the (local) maximum of their values; then leaf processes pass their local maxima to their parents and halt while each interior process, after waiting for the arrival of maxima from its two descendants, compares these two values with its local maximum and passes the largest of the three to its parent; the (global) maximum is ejected by the root.

These examples are not intended to have any particular form from which specificational mechanisms might be inferred; in fact their description has been compressed and restated from the original. They are intended only as informal statements of the essential aspects of two "typical" algorithms to be used to illustrate our points.

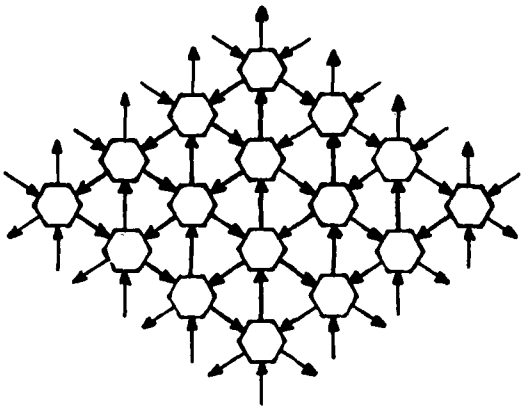


Figure 1. Kung-Leiserson band-matrix multiplication algorithm.

We now identify five characteristics commonly exhibited by the descriptions of parallel algorithms for the nonshared memory model of parallel computation:

- a *graph*, G = (V,E), whose vertex set, V, represents processors, and whose edge set, E, represents the communication structure of the algorithm,
- a *process set*, P, describing the types of computational activity to be found in the algorithm,
- an *assignment* function, π: V $\longrightarrow$ P giving to each processor a process,
- a *synchronization statement* describing the interaction of the separate computational elements,
- an *input/output statement* describing the assumed form of the data, and the format of the results

There is nothing surprising about the entries on this list. They arise all the time in parallel algorithm descriptions, which is exactly the point. Let us see how they were used in the Examples.

In the case of the band-matrix multiplication algorithm, the graph *G* is given in Figure 1.* For the maximum-finding algorithm, the graph is a complete binary tree.

*Strictly speaking, this is not a graph, but we intend that the edges around the perimeter be connected to input/output "vertices."

It is significant that the information describing the communication structure of the algorithm was, or could have been, given by a picture. Notice that in both cases the graph is really a *representative* for a graph *family*. Problems of different input size will require graphs of different size; for example, the structure in Figure 1 is appropriate for matrices with bands that are four values wide, and matrices with bands of width five would require 25 processors. The way in which the graph family is defined, the way in which a particular input size determines a particular graph family member, and the way to handle the cases where the graph has more vertices than the available number of processors, are examples of "commonly omitted details," cited above as the first source of dissimilarity. They are omitted because they are obvious, or their effect is inconsequential, or they are irrelevant. The graph, by contrast, is fundamental, as is the fact that its size is determined by the problem's input size.

The process set P for the systolic algorithm contains three elements: one with the inner product step first followed by the two idle steps, one with the inner product between the idles, and one with it following the idles. The maximum finding algorithm has two elements in its process set: an interior node process that receives descendant inputs, and a leaf process that does not. Notice that the size of the process set is fixed, independent of the size of the graph.

The assignment function π is given in the case of the systolic array by describing which horizontal rows are performing the inner product step and which are idling. The fact that processing is performed on every third row together with the fact that the outputs of an inner product step are obviously transmitted to processes that will read the values on the next step is sufficient information to assign the processes to processors. For the binary tree algorithm the assignment is implicit in the use of the phrases "leaf process" and "interior process," i.e. we know which nodes are leaf nodes and they obviously get leaf processes.

The synchronization specification may not be very recognizable in either algorithm, but it is there. The processes of the systolic algorithm operate in lock step, since the time of an idle step equals the time of an inner product step. This means that the question of when processes read and write is determined in this case simply by the explicit use of the idle. In the tree algorithm the phrase "after waiting for the arrival" says that the process operation is "data-driven," i.e. the process executes until it needs data and then it idles until the data has been read.

The input/output statement is curious in that it must be known to program the process set, it can influence the synchronization, and it is critical to demonstrating the correctness of the algorithm, but it seems only to enter the process explicitly after the program is written and is ready to be run. Since its indirect influence permeates the programming activity and is hard to extract, we will mention input/output little after these following two points. First, the input/output for the band-matrix product algorithm was given pictorially in the original description,[5] offering another example of the role of pictures. Second, the data in both examples can be viewed as streams, i.e. it is unstructured.

Although there might be other characteristics commonly exhibited by parallel algorithm specifications, this set of five properties is sufficiently comprehensive to serve as our "standard algorithmic form." It summarizes the things that computer scientists describe about an algorithm when they explain it to each other. Making the program form of our parallel programming environment close to this standard algorithmic form will contribute to reducing dissimilarity due to mechanism mismatch. We next consider the extent to which this has been achieved in the Poker environment.

The Mechanisms of Poker

The statement that the communication structure of a parallel algorithm can be, and frequently is, given as a graph is simply a comment on the nature of computation in the nonshared memory model of parallelism. It does not, by itself, indicate a convenient mechanism for expressing such a graph. The graph could be given by a pointer structure or by an adjacency matrix,[7] it could be defined implicitly by a routine that computes packet addresses, or in any of a dozen other ways. The choice will affect the degree of mechanism mismatch between program and algorithm, of course, and again one should be guided by what is actually used in the literature. The mechanisms selected for the Poker system represent one way to balance the convenience of "user friendly" mechanisms and the pragmatics of efficient implementation.

The first property for which a mechanism must be selected is the graph used to define the communication structure of the algorithm. The most convenient way to express a graph is evidently with a picture, judging from how frequently they are used to describe graphs. Thus, the Poker Environment provides an interactive graphics system as the mechanism for defining the graph. The graph is not drawn free-form, but rather it is laid out on a stylized two-dimensional medium called a

lattice (see Figure 2). In the lattice, squares represent potential vertex positions that can be connected by line segments to define edges. The circles provide sites where line segments can bend and crossover one another. The line segments are drawn by moving a cursor from circle to circle along the path of the intended edge. Such a drawing is called an *embedding* of the graph into the lattice.

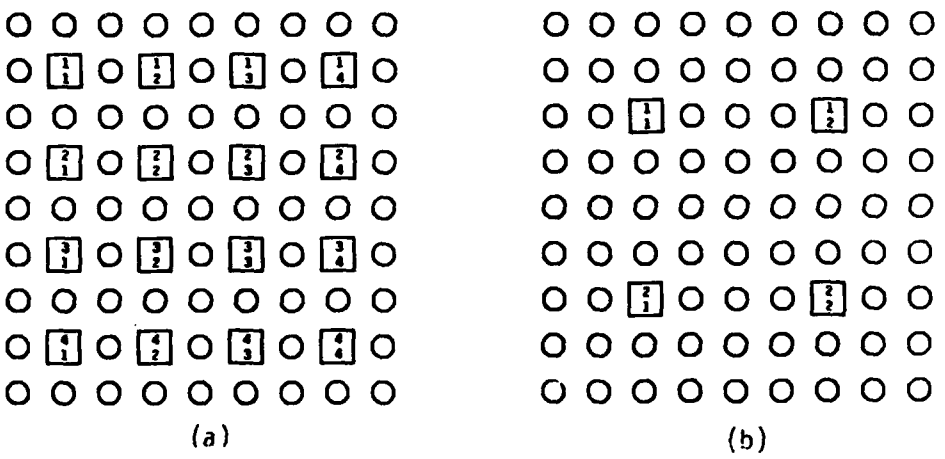


Figure 2. Lattices for graph embedding.

Figure 3 shows how to program the communication structures for the graphs of the example algorithms. The hexagonal mesh is, except for a 45° rotation, a direct implementation of Figure 1. The binary tree must be deformed to fit in the particular lattice medium, but the embedding is still a mechanical process in general.[4]

Notice that the embedding activity is graphical programming rather than symbolic programming. Since graphs tend to be given graphically rather than symbolically, this probably represents a reduction is dissimilarity over the symbolic alternatives.

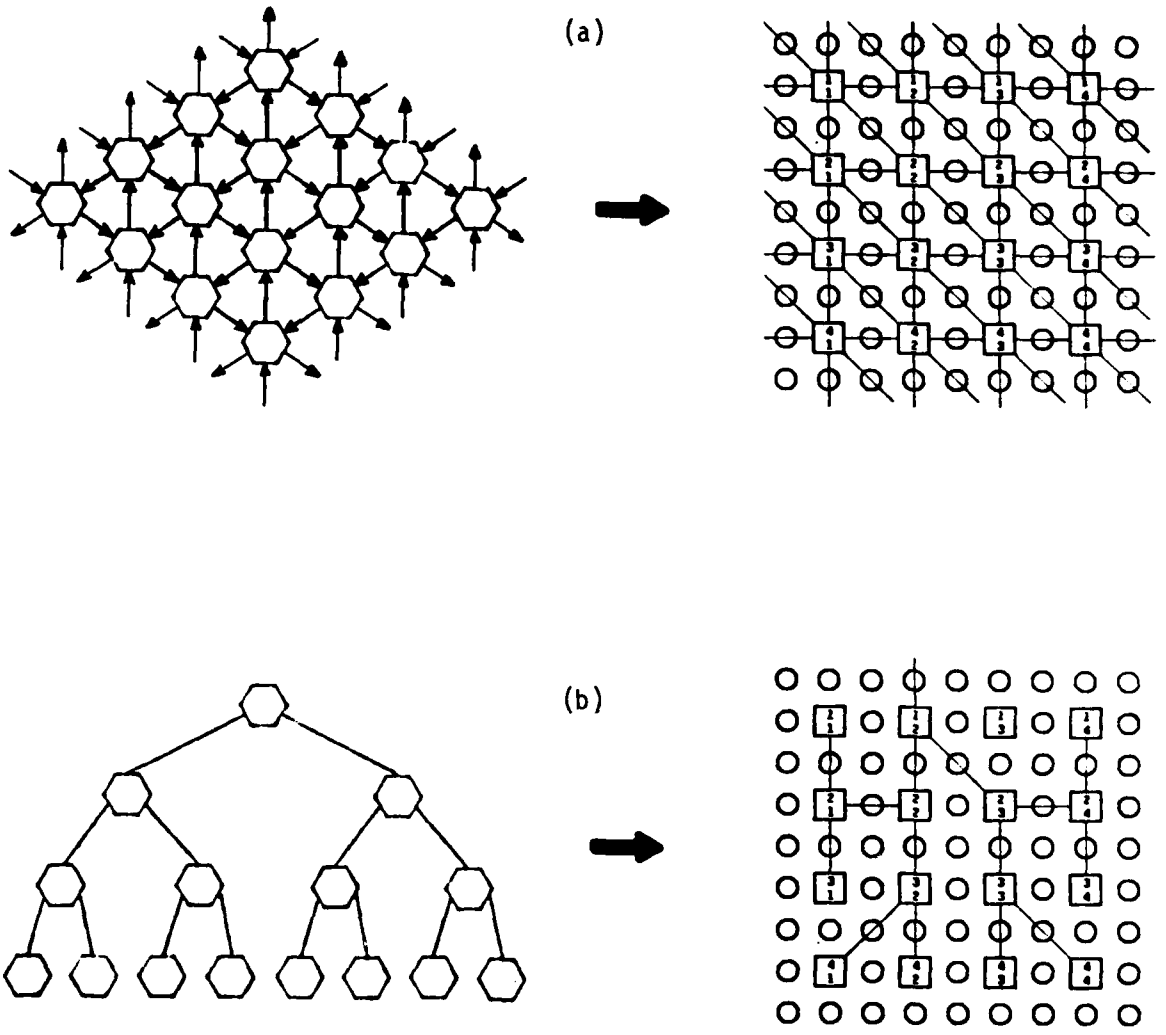


Figure 3. Embedding of the graphs describing the communication structures for algorithms of Examples I and II.

The next step is to specify the sequential code segments of the processes, i.e., to define the elements of the process set *P*. This is conventional symbolic programming and could be done in any sequential language, e.g. C, Pascal, etc. The Poker Environment uses a primitive language, called *XX*, for this purpose, and the mechanism is to define a set of independent procedures. Figure 4 shows two sample process codes.

```
code inner;
ports Ain, Aout, Bin, Cin, Cout;
begin
  real A, B, C;
  while true do
  begin
  A ← Ain; B ← Bin; C ← Cin;
  C := C + A * B;
  Aout ← A; Bout ← B; Cout ← C;
  end
end
```

(a)

```
code inode (logofn);
ports lchild, rchild, parent;
begin
  integer logofn, i;
  real big, temp, vals[logofn];
  big := vals[1];
  for i := 2 to logofn do
    if big < vals[i] then big := vals[i];
  temp ← lchild;
  if big < temp then big := temp;
  temp ← rchild;
  if big < temp then big := temp;
  parent ← big
end
```

(b)

Figure 4. The XX code for processes from Examples I and II.

When compared with the logical process presented in Example I, the code shown in Figure 4a is seen to differ in two ways. First, there are the syntactic features, key words, declarations, etc., characteristic of standard programming languages; second, there is explicit mention of "ports." Ports provide a means for a process to refer to the processes with which it communicates: To read from another process, it assigns from a port name (e.g., A ← Ain) and to write to another process it assigns to a port name (e.g., Aout ← A). Which processes these will be and how they are specified cannot be addressed until port names are defined, below.

The mechanisms for specifying the assignment of a process to each vertex, $\pi: V \rightarrow P$, is to display the graph embedding and request that the user assign process names (the name following code in the procedure definition) to each vertex. Figure 5 shows the assignment for the example programs. Notice that the actual parameter for the tree program is given on the line following the name.

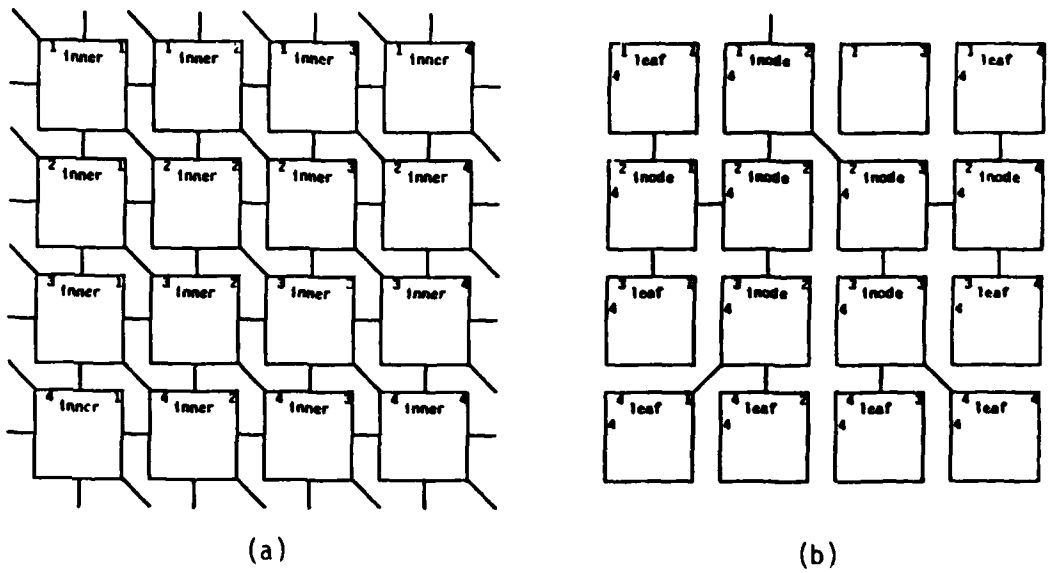


Figure 5. Process assignment for the examples.

For the band-matrix product systolic array, one might expect three different process types in the set P, one corresponding to each of the three positions of the inner product step in a pair of idles, as described in Example I. But the XX code of Figure 4 does not have any idle instructions. This is because the XX programming language uses a data-driven semantics to define the synchronization of processes: the reads wait for the arrival of the data. The three process types mentioned in Example I reduce to one type when data-driven semantics are used. How is synchronous communication achieved in Poker? Automatically. After the whole program specification is complete, a code optimization facility, called *coordination*, analyzes the program and converts it to a synchronously communicating program, if possible. In this respect programming an algorithm with Poker is somewhat easier than defining the algorithm in the first place.

It must be emphasized that not all Poker programs can be efficiently coordinated. XX has an idle instruction for those cases when synchronous execution is desired, but automatic coordination is not possible. The user could always decide to do his own coordination, but this is probably not advisable since coordination, like code optimization, is better done by machine. For example, automatic coordination can produce a better algorithm than the one with the explicit idles described in Example I.[8]

With four of the mechanisms defined, the programming is nearly complete. All that remains is to interface the graphical communication structure with the symbolic code segments. This is accomplished by a mechanism that assigns port names to the edges of the graph that meet at a vertex so that the process assigned to that vertex can refer to its logical neighbors. The port name assignment is a feature of Poker programming with no direct analog in the conceptual discussion of parallel programming given above. It can be viewed as an explicit means of establishing a vocabulary with which to describe communication. Terms like "parent" and "left neighbor" have no intrinsic meaning; they must be specified to the computer. Thus, port naming is a type of declaration.

At this point the Poker program is finished. It can now be compiled, coordinated, assembled and linked. To run the program we must give the input/output. This is done with a mechanism that labels the edges connecting to the perimeter of the lattice with the names of the streams that flow in or out of them. In this way the edge edges are ports to the lattice. The data, of course, moves to and from the lattice in a data-driven manner.

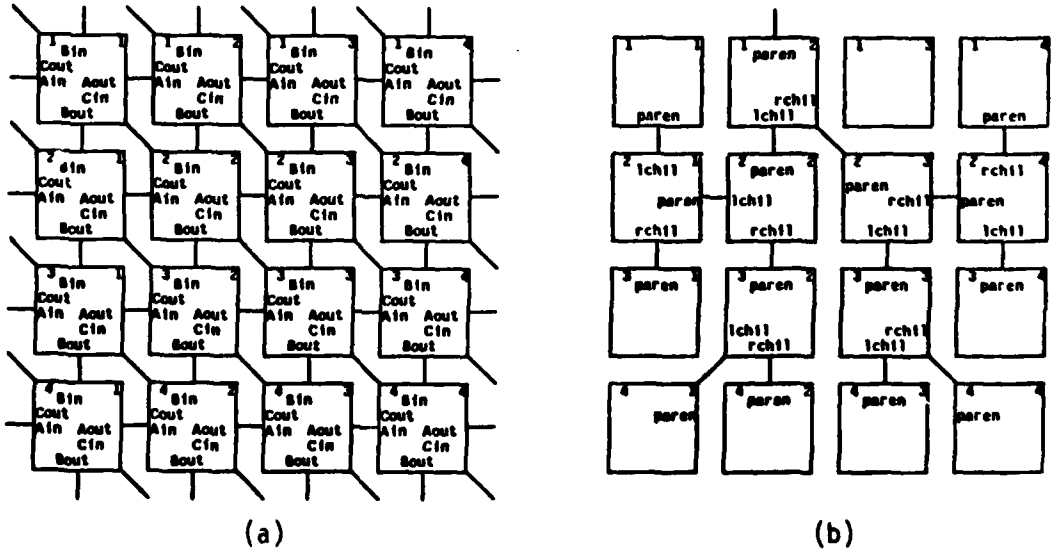


Figure 6. Port names declaration for the examples; note that names are clipped to the first five characters.

The mechanisms supplied by Poker to program an algorithm are: drawing a picture of a graph representing the communication structure, defining a set of sequen-

tial procedures, labeling a picture of the graph with process names in one case and port names in the other, and data-driven semantics with coordination for synchronization; to run the program the I/O is specified as named streams of data. There is one other mechanism provided by Poker that has not been mentioned, the phase construction.

Informally, a "phase" is an algorithm (in the sense we've been using) with a single communication graph. Each of our example algorithms is a phase, since each is based on a single communication graph. Algorithms found in the technical literature frequently qualify as phases. (See the Thompson and Kung[9] sorting algorithm for an example of a multiphase, i.e. multiple interconnection structure, algorithm.) "Real World" problems -- the complicated, ill-defined, exception prone programming situations that application programmers will be solving when they use the programming environment -- tend not to be so tidy. Solutions to real world problems will presumably be developed by dividing them into parts which can either be solved directly by a phase, or further subdivided until its constituents can be solved by phases. The consequence of this strategy is that phases must be composed, i.e. put together to form more complicated algorithms. For example, the conjugate gradient method of solving partial differential equations can be done with four phases, an input phase, a "grid" phase for matrix multiplication, a "tree" phase for summation and broadcast, and an output phase[14]. The "grid" and "tree" phases are executed iteratively in an alternating schedule. Poker has been designed to support phases and their composition.

Description of the Poker Environment

After considerable discussion of various concepts and abstractions, it is time to get down to the details of the Poker system. Perhaps the first question to be asked following the discussion of the Poker mechanisms is, what does a whole Poker program look like? Answer: It cannot be seen *in toto*. Unlike "regular" programs, Poker programs are not monolithic pieces of program text. Instead, a Poker program is a database. To see the communication structure, one displays a picture of the graph which is stored in the database. To see the assignment function, one displays a picture of the graph labeled with process names.** To make changes to the program

**This picture is not actually stored directly in the database -- it is constructed by the Poker system from the database relations. How this is done is interesting, but beyond the scope of this paper; see Ref. 10.

one simply changes the picture or the labeling which causes the database to be changed. But we are ahead of ourselves; let's go back and start at the beginning of a Poker session.

The Poker environment uses two displays, primary and secondary, one of which must be a high resolution (1024 x 768 pixel) bit-mapped display. Two displays are used simply to increase the amount of information available to the programmer at any one time. Most activity takes place on the primary display; XX programming is usually done on the secondary display.

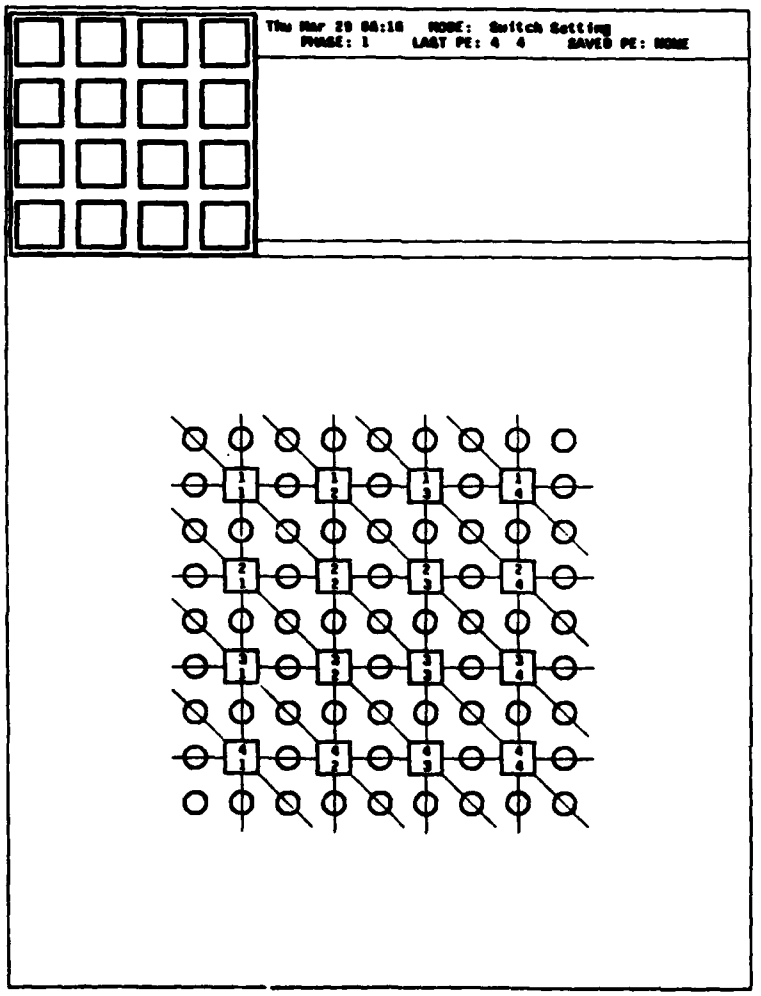


Figure 7. The form of a typical primary display, showing a 16 processor switch setting display.

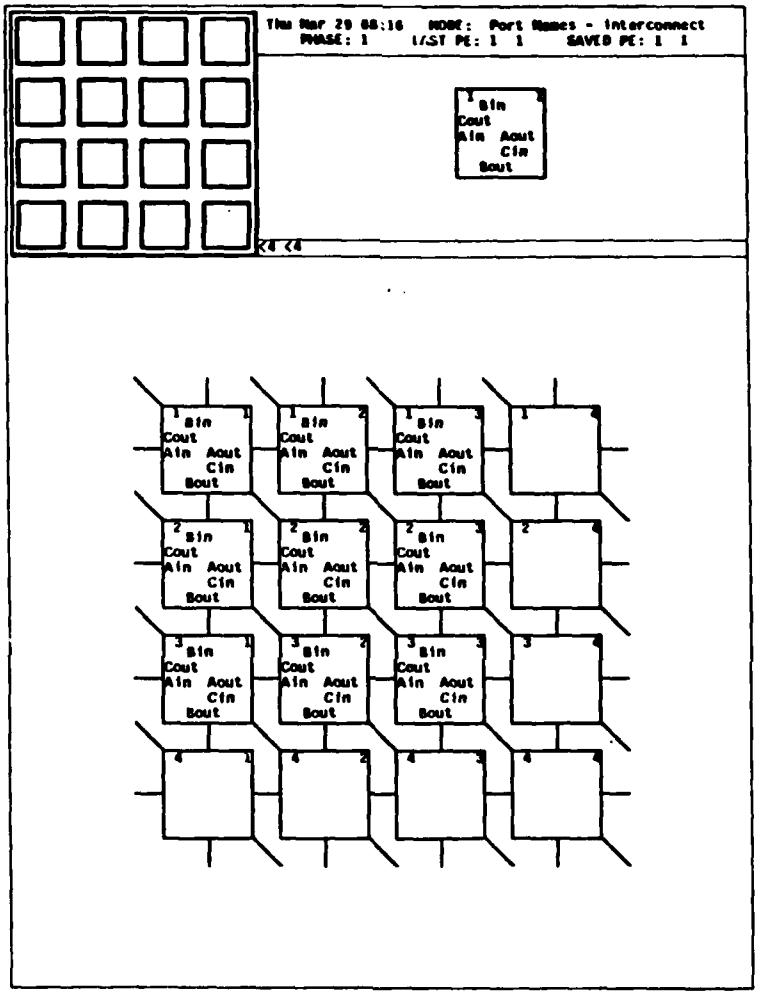


Figure 8. A 16 processor port names display.

The primary display has the form illustrated in Figure 7. The bottom square region, called the *field*, is where most of the programming activity takes place. The field always displays some schematic representation of the lattice being programmed. The exact form of the representation changes (compare Figure 7 with Figure 8) depending on whether the programmer is performing a graph embedding, a process assignment, a port declaration, etc. Status information, diagnostics and miscellaneous data are given in the upper right region of the display, called the *chalkboard*. (The upper left region gives a map of the lattice marked with that portion being displayed in the field; this is useful only for problems larger than shown in Figure 7.)

The bottom line of the chalkboard is the *command line*, used for specifying the few textual commands required by Poker,*** such as reading library files.

The logical structure of the Poker Environment is shown in Figure 9. It provides an integrated set of facilities to

- define architecture characteristics (CHiP Parameters),
- embed communication graphs (Switch Settings),
- program process set codes, (XX Language),
- declare port names (Port Naming),
- assign processes to processors (Code Naming),
- compile, coordinate, assemble, load and define input/output (Command Request),
- execute, trace, peek and poke (Trace Values).

Although the facilities have been listed in the order in which they might be used when writing a program, notice from Figure 9 that no order is actually enforced; programmers can, and typically do, jump back and forth between the different facilities.

***Poker is extremely interactive; most actions are given as a single key stroke and have immediate effect.

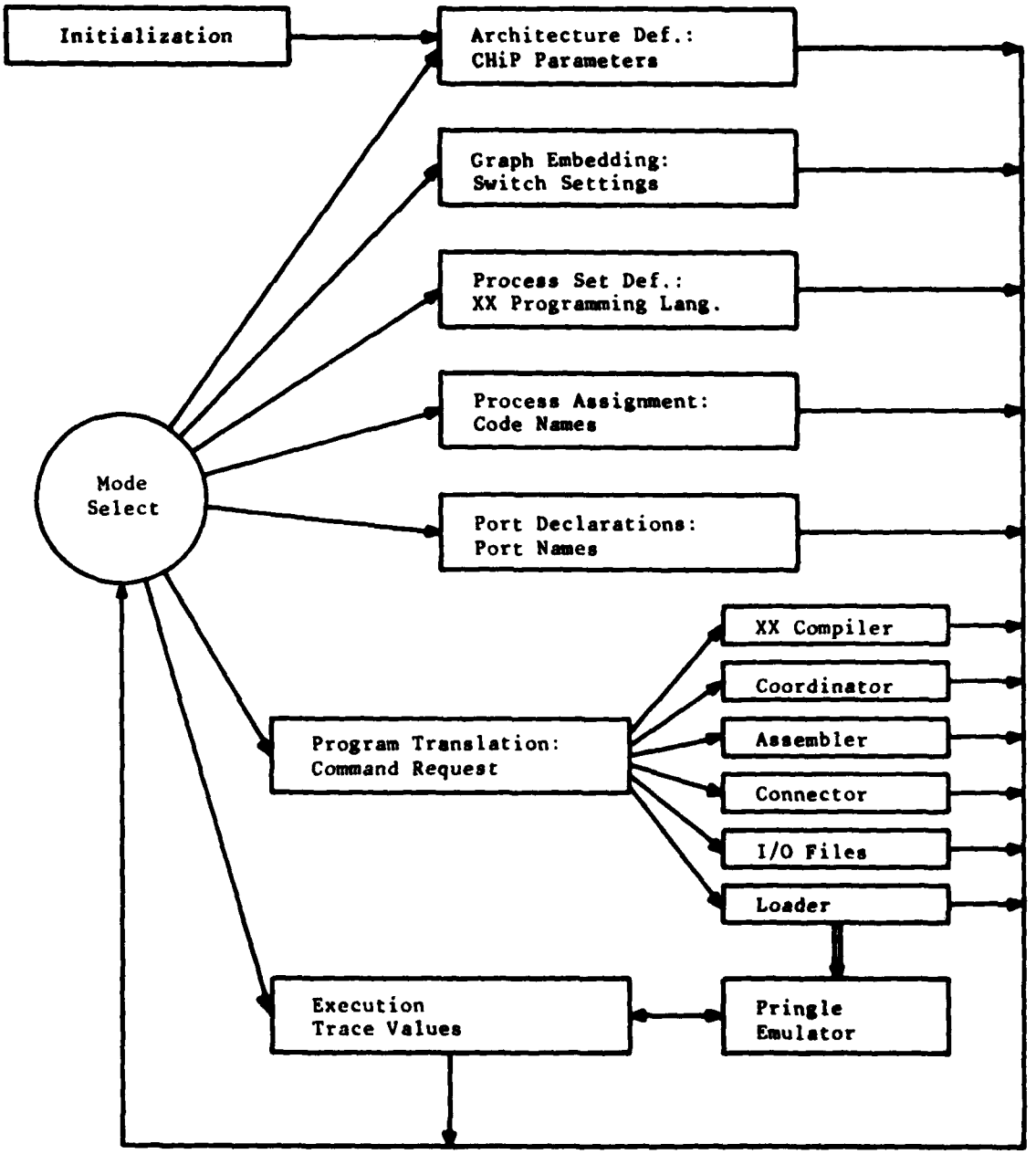


Figure 9. The logical structure of the Poker environment.

Next we briefly describe the kinds of information displayed with each facility, and the service provided. The reader should be cautioned that much is being left out in the interest of brevity, though full details are available,[11] and that the dynamic, interactive character of the system is completely lost in this hard copy presentation.

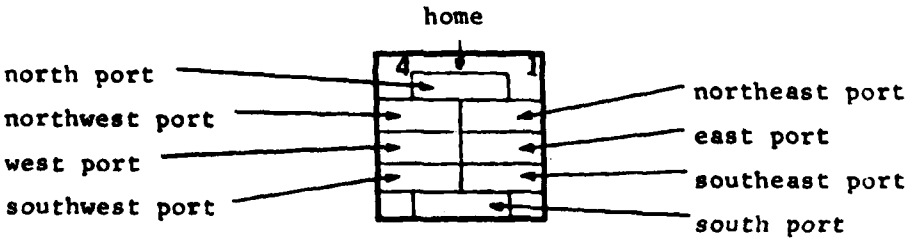
Architectural definition. Because Poker is intended to support CHiP programming, it has been designed to accommodate a number of CHiP family architectures. Programs can be written for logical CHiP machines with from 4 to 4096 processors. All of these logical machines can be emulated using a software emulator. (The emulated instruction set is that of the Pringle parallel computer,[12] a hardware emulator of the 64 processor members of the CHiP family.) Consequently, the programmer begins using Poker by specifying the characteristics of the underlying logical architecture. These include the number of processing elements and the amount of routing capability needed for the lattice (corridor width[4]). The default parameters are those that match the machine defined in the previous session, or, if there was none, then the parameters of the Pringle hardware.

Embedding communication graphs. The field of the primary display shows the lattice of the current architecture, as illustrated in Figure 2. The programmer defines the communication structure of the algorithm by drawing the graph in the lattice that defines the structure. This chiefly involves connecting processors (represented as boxes) with line segments to define edges. Graphics primitives, based on cursor keys, permit edges to be drawn and erased. Facilities are available for managing the display, saving embeddings, reading in library embeddings, etc.

Programming the process set codes. The XX sequential programming language is a simple scalar language for defining processes. The language has four data types (Boolean, character, integer and real), the common control structures (**while**, **for**, **if-then-else**, etc.), vectors and the usual supply of scalar arithmetic and logical operators. In addition to data type declarations, one can also declare scalar variables to be port names, procedure parameters, or variables to be traced. Input/output is performed by assigning from or to a port name. The semantics are "data-driven": writes occur immediately and reads wait on the arrival of data, if necessary. XX process codes are generally developed on the secondary display using a standard editor.

Process assignment. The processors are assigned processes using a field display on the primary terminal like the two shown in Figure 5. The programmer enters the name of the process procedure on the first line of the box symbolizing the processor. If the process has formal parameters, then values for the actual parameters can be entered on the following (four) lines. For example, the formal parameter *logofn* in Figure 4b is assigned the value 4 in Figure 5b. Facilities are provided to avoid tedious typing. One can buffer the contents of a box and then automatically deposit the contents of the buffer into processors in whole regions of the processor array. For example, if the programmer buffers a box, then typing *<4 <4* followed by the insert key causes the processors whose indices are both less than 4 to receive the contents of the box. (The same mechanism, but for port declarations, is shown in Figure 8.) Standard screen management facilities, library access facilities, etc. are available.

Port declarations. The field of the primary display has the form like the two examples shown in Figure 6. Each processor has up to eight incident edges as a result of the graph embedding, and it has been assigned a process which refers to up to eight port names. These are matched using the port declaration. The processor box is divided into eight windows:



The programmer enters the names used by the assigned process code into the window for that edge. The names are clipped to the first five characters. Facilities are provided for displaying unclipped names in the chalkboard. Like the process assignment, it is possible to buffer port assignments and deposit them automatically in whole regions of the processor array (Figure 8). Screen management and other ancillary commands are available.

Program translation. The preceding facilities provide a means of specifying a Poker database containing the elements of a parallel program. They are then converted into executable form. The XX compiler converts each process into assembly code. The coordinator[13] then attempts to convert the process assigned to each

processor into a form that permits the entire program to run with synchronous (i.e., not data-driven) execution. (This step can be by-passed and the processes can be run in data-driven form.) If coordination is successful, the processors may all have different assembly codes associated with them. In any event object code is produced. The connector "compiles" the graphical representation of the communication graph into a symbolic object form. The object code and the object graph as well as the actual parameter values are loaded into the emulator (or the Pringle). Finally, the input/output files are specified.

Execution. The resulting program is executed. The traced variables are displayed in a field similiar to that used for process assignment. The execution can proceed for a given number of steps, or until a displayed value changes. When the execution is suspended, any of the displayed values can be changed. When execution resumes these new values are poked (whence the name Poker) back into the processor memories.

The Poker system has been implemented as a (~40,000 line) C program to run on a VAX 11/780 under the UNIX (a trademark of Bell Laboratories) operating system.

Program Performance

How do programs written in the Poker environment perform on the CHiP Computer? Since there is no CHiP Computer implementation and since there is scant experience with the Pringle, it is not possible to support our claims with copious evidence. But claims can be made nevertheless.

Generally speaking, Poker introduces little inefficiency. Its graphical programming facilities -- switch settings, port names and code names -- engender no inefficiency. The latter two are only definitional. The switch settings are directly translated into source-target pairs for the Pringle and will be *literally* translated for the CHiP Computer. The XX language is so simple that very efficient compilation of PE code is possible. (The language could be richer and still be efficient; replacement with another sequential language, e.g. Pascal, is a simple change.) There is only one XX feature with noticeable execution time inefficiency, the data-driven input/output, and even this is only occasionally a liability.

Data-driven I/O is both a luxury and a necessity. It is a luxury in that certain programs, capable of being run synchronously, need not be written using the tedious process of inserting explicit idles: They can be written with data-driven communication and then be run through the coordination phase[13] to be converted into synchronous form. Some programs, however, cannot be coordinated. Others cannot be run synchronously without introducing superfluous I/O, or "chattering", in which the processors communicate back and forth at regular intervals whether or not there is actually any data to be sent.[13] These programs ought to be written using data-driven communication because it is easier and substantially more efficient than the "chattering". In such cases data-driven I/O is a necessity. To the extent that a program *could be* run with synchronous I/O but is not, either because it is not coordinated at all or the coordinator fails to find a synchronous variant, there is a small loss in performance. As an example of the case where coordination is not used, we know that the Kung-Leiserson band-matrix product algorithm[5] takes 1.16 times longer using uncoordinated data-driven rather than coordinated data-driven communication.[8] Since data-driven I/O is necessary for the nonsynchronously executable programs anyway, the inefficiency arises only when the coordinator phase fails to find a synchronous variant and one exists. This is analagous to criticizing sequential languages because their optimizers occasionally fail to find an optimization.

Thus, the Poker environment is a very efficient programming system for the CHiP Computer. But how well does it support other ensemble machines?[15]

Since the PEs of the CHiP Computer and the Pringle are similar to other ensemble machines and an XX compiler for them would be similarly efficient, the inefficiencies will arise in expressing the communication structure. Postulate an ensemble machine with a fixed interconnection structure S and consider using Poker to program such a machine. There are two possibilities. First, one can configure the lattice once and for all time to be the interconnection structure S. Then if the algorithm uses a different communication structure than S, it is up to the programmer to encode the appropriate routing actions in the PE processes. In this case the burden of mapping the communication structure onto the architecture is entirely on the programmer. Poker would be of little help. Second the programmer could use Poker switch settings to express the algorithm's communication structure just as if the target machine was the CHiP Computer. Then if that structure did not match S, either an automatic or a manual scheme for embedding the graph into S could be used. This might take the form of packet address encoding if that would be ap-

propriate for the architecture. The advantage would be that the interconnection graph mechanism would still be a convenience for the programmer. The disadvantage is that there could be inefficiencies in the run time implementation of the processor-to-processor communication, but this would be due to the architecture, not Poker.

Summary

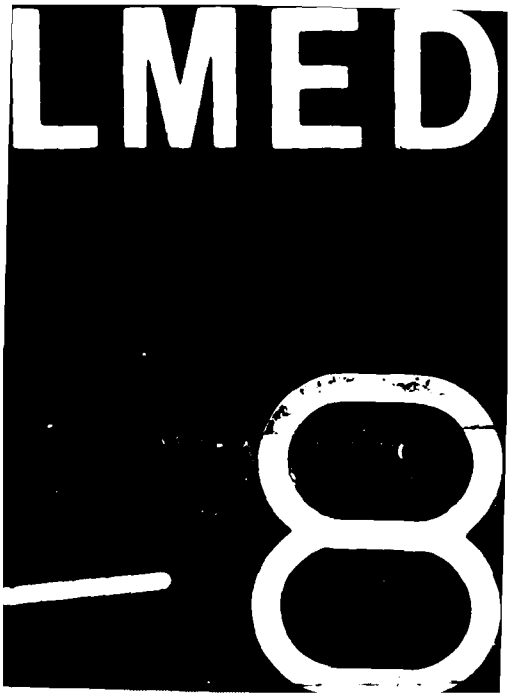
Starting from first principles we have descended from a "high level" abstract idea of what is required for parallel programming to the basic details of a particular parallel programming environment. Although each step got more specific and closer to the realities of everyday parallel programming, we kept in sight our original observation that parallel algorithms often utilize five common properties of their specification: a graph describing the communication structure, a finite process set defining the activities, an assignment function giving processes to processors, a synchronization statement and input/output information. These properties motivated mechanisms, and the mechanisms were illustrated by the Poker environment.

Acknowledgements

The Poker Environment represents the ideas and hard work of many people. Dennis B. Gannon and Janice E. Cuny contributed to many of the concepts as well as to the design of XX. Version 1.0 of Poker was implemented almost entirely in the summer of 1982 at Purdue University by a delightful and committed group of gentlemen, Steven S. Albert, Carl W. Amport, Brian B. Beuning, Alan J. Chester, John P. Guaragno, Christopher A. Kent, John Thomas Love, Eugene J. Shekita and Carleton A. Smith, and significant enhancements have since been made by Steven J. Holms. The presentation of Poker in this paper benefitted from helpful comments by the referees. All of this help is deeply appreciated.

References

1. R.H. Perrott and D.K. Stevenson, "User's Experience with the ILLIAC IV System and Its Programming Languages," *SIGPLAN Notices*, Vol. 16, No. 7, July, 1981, pp. 75-81.
2. Peter B. Lax, *Report of the Panel on Large Scale Computing for Science and Engineering*, National Science Foundation, GPO, 1982.
3. Robert F. Cotellessa, *Report of the Information Technology Workshop*, National Science Foundation, GPO, 1983.
4. Lawrence Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer*, Vol. 15, No. 1, January, 1982, pp. 47-56.
5. H.T. Kung and Charles E. Leiserson, "Algorithms for VLSI Processor Arrays," in Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.
6. J.T. Schwartz, "Ultracomputers," *TOPLAS*, Vol. 2, No. 4, October, 1980, pp. 484-521.
7. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Comptuer Algorithms*, Addison-Wesley, 1974.
8. Janice E. Cuny, Working Notes on Buffer Size, unpublished Blue CHiP Project Notes, August 1983.
9. C.D. Thompson and H.T. Kung, "Sorting on a Mesh-Connected Parallel Computer, *CACM*, Vol. 20, No. 4, April, 1977, pp. 263-271.
10. Lawrence Snyder, Steven S. Albert, Carl W. Amport, Brian G. Beuning, Alan J. Chester, John P. Guaragno, Christopher A. Kent, John Thomas Love, Eugene J. Shekita, Carleton A. Smith, "The Poker Programming Environment and its Implementation," Technical Report CSD-TR-410, Purdue University, 1982.
11. Lawrence Snyder, "Poker (1.0) Programmers Guide," Technical Report CSD-TR-434, Purdue University, 1983.
12. Alejandro A. Kapanan, J. Timothy Field, Dennis B. Gannon and Lawrence Snyder, "The Pringle Parallel Computer," in *Proceedings of the 11th International Symposium on Computer Architecture*, 1984.
13. Janice E. Cuny and Lawrence Snyder, "Conversion from Data Driven Programs for Synchronous Execution," in *Proceedings of the 10th POPL*, ACM, 1983, pp. 197-202.
14. Dennis B. Gannon, Lawrence Snyder and John Van Rosendale, "Programming Substructure Computations for Elliptic Problems on the CHiP System," in Ahmed K. Noor (ed.), *Impact of New Computing Systems on Computational Mechanics*, The American Society of Mechanical Engineers, 1983, pp. 65-80.
15. Charles E. Seitz, "Ensemble Architectures for VLSI -- A Survey and Taxonomy" in Paul Penfield (ed.), *Proceedings of the Conference on Advanced Research in VLSI*, Artech House, 1981, pp. 130-135.


LMED
8